Quick Order Fairness

Christian Cachin[1 [0000−0001−8967−9213]], Jovana Mićić[1 [0000−0002−5308−9155]], and Nathalie Steinhauer[1 [0000−0001−5029−7656]]

Institute of Computer Science, University of Bern, Switzerland
firstname.lastname@inf.unibe.ch

Abstract. Leader-based protocols for consensus, i.e. atomic broadcast, allow some processes to unilaterally affect the final order of transactions. This has become a problem for blockchain networks and decentralized finance because it facilitates front-running and other attacks. To address this, *order fairness* for payload messages has been introduced recently as a new safety property for atomic broadcast complementing traditional *agreement* and *liveness*. We relate order fairness to the standard validity notions for consensus protocols and highlight some limitations with the existing formalization. Based on this, we introduce a new *differential* order-fairness property that fixes these issues. We also present the *quick order-fair atomic broadcast protocol* that guarantees payload message delivery in a differentially fair order and is much more efficient than existing order-fair consensus protocols. It works for asynchronous and for eventually synchronous networks with optimal resilience, tolerating corruptions of up to one third of the processes. Previous solutions required there to be less than one fourth of faults. Furthermore, our protocol incurs only quadratic cost, in terms of amortized message complexity per delivered payload.

Keywords: consensus · atomic broadcast · decentralized finance · front-running attacks · differential order fairness.

1 Introduction

The nascent field of *decentralized finance* (or simply *DeFi*) suffers from insider attacks: Malicious miners in permissionless blockchain networks or Byzantine leaders in permissioned atomic broadcast protocols have the power of selecting messages that go into the ledger and determining their final order. Selfish participants may also insert their own, fraudulent transactions and thereby extract value from the network and its innocent users. For instance, a decentralized exchange can be exploited by *front-running*, where a genuine message m carrying an exchange transaction is *sandwiched* between a message m_{before} and a message m_{after}. If m buys a particular asset, the insider acquires it as well using m_{before} and sells it again with m_{after}, typically at a higher price. Such front-running and other price-manipulation attacks represent a serious threat. They are prohibited in traditional finance systems with centralized oversight but must be prevented

technically in DeFi. Daian *et al.* [7] have coined the term *miner extractable value (MEV)* for the profit that can be gained from such arbitrage opportunities.

The traditional properties of *atomic broadcast*, often somewhat imprecisely called *consensus* as well, guarantee a total order: that all correct parties obtain the same sequence of messages and that any message submitted to the network by a client is delivered in a reasonable lapse of time. However, these properties do not further constrain *which* order is chosen, and malicious parties in the protocol may therefore manipulate the order or insert their own messages to their benefit. Kelkar *et al.* [14] have recently introduced the new safety property of *order fairness* that addresses this in the Byzantine model. Kursawe [15] and Zhang *et al.* [19] have formalized this problem as well and found different ways to tackle it, relying on somewhat stronger assumptions.

Intuitively, *order fairness* aims at ensuring that messages received by "many" parties are scheduled and delivered earlier than messages received by "few" parties. The *Condorcet paradox* demonstrates, however, that such preference votes can easily lead to cycles even if the individual votes of majorities are not circular. The solution offered through *order fairness* [14] may therefore output multiple messages *together as a set* (or batch), such that there is *no order* among all messages in the same set. Kelkar *et al.* [14] name this property *block-order fairness* but calling such a set a "block" may easily lead to confusion with the low-level blocks in mining-based protocols.

In this paper, we investigate order fairness in networks with n processes of which f are faulty, for asynchronous and eventually synchronous atomic broadcast. This covers the vast majority of relevant applications, since timed protocols that assume synchronous clocks and permanently bounded message delays have largely been abandoned in this space.

We first revisit the notion of block-order fairness [14]. In our interpretation, this requires that when n correct processes broadcast two payload messages m and m', and $\overline{\gamma}n$ of them broadcast m before m' for some $\overline{\gamma} > \frac{1}{2}$, then m' is not delivered by the protocol before m, although both messages may be output together. This guarantee is difficult to achieve in practice because Kelkar *et al.* [14] show that for the relevant values of $\overline{\gamma}$ approaching one half, the resilience of any protocol decreases. Tolerating only a small number of faulty parties seems prohibitive in realistic settings.

More importantly, we show that $\overline{\gamma}$ cannot be too close to $\frac{1}{2}$ because $\overline{\gamma} \geq \frac{1}{2} + \frac{f}{n-f}$ is necessary for any protocol. This result follows from establishing a link to the differential validity notion of consensus, formalized by Fitzi and Garay [10]. Notice that block-order fairness is a relative measure. We are convinced that a differential notion is better suited to address the problem. We, therefore, overcome this inherent limitation of relative order fairness by introducing *differential order fairness*: When the number of correct processes that broadcast a message m before a message m' exceeds the number that broadcast m' before m by more than $2f + \kappa$, for some $\kappa \geq 0$, then the protocol must *not* deliver m' before m (but they may be delivered together). This notion takes into account existing results on differential validity for consensus [10]. In particular,

when the difference between how many processes prefer one of m and m' over the other is smaller than $2f$, then *no protocol exists* to deliver them in fair order.

Last but not least, we introduce a new protocol, called *quick order-fair atomic broadcast*, that implements differential order fairness and is much more efficient than the previously existing algorithms. In particular, it works with optimal resilience $n > 3f$, requires $O(n^2)$ messages to deliver one payload on average and needs $O(n^2L + n^3\lambda)$ bits of communication, with payloads of up to L bits and cryptographic λ-bit signatures. This holds for *any* order-fairness parameter κ. For comparison, the asynchronous Aequitas protocol [14] has resilience $n > 4f$ or worse, depending on its order-fairness parameter, and needs $O(n^4)$ messages.

To summarize, the contributions of this paper are as follows:

- It illustrates some *limitations* that are inherent in the notion of block-order fairness (Section 4.1).
- It introduces *differential order fairness* as a measure for defining fair order in atomic broadcast protocols (Section 4.2).
- It presents the *quick order-fair atomic broadcast protocol* for differentially order-fair Byzantine atomic broadcast with optimal resilience $n > 3f$ (Section 5).
- It demonstrates that the quick order-fairness protocol has quadratic amortized message complexity, which is an n^2-fold improvement compared to the most efficient previous protocol for the same task (Section 6).

For lack of space, some material, proofs and detailed protocol analysis are omitted here, but available in the full version [6].

2 Related work

Over the last decades, extensive research efforts have explored the state-machine replication problem. A large number of papers refer to this problem, but only a few of them consider fairness in the order of delivered payload messages. In this section, we review the related work on fairness.

Kelkar *et al.* [14] introduce a new property called *transaction order-fairness* which prevents adversarial manipulation of the ordering of transactions, i.e., payload messages. They investigate assumptions needed for achieving this property in a permissioned setting and formulate a new class of consensus protocols, called Aequitas, that satisfy order fairness. A subsequent paper by Kelkar *et al.* [12] extends this approach to a permissionless setting. Recently, Kelkar *et al.* [13] presented another permissioned Byzantine atomic-broadcast protocol called Themis. It introduces a new technique called *deferred ordering*, which overcomes a liveness problem of the Aequitas protocols.

Kursawe [15] and Zhang *et al.* [19] have independently postulated alternative definitions of order fairness, called *timed order fairness* and *ordering linearizability*, respectively. Both notions are strictly weaker than order fairness of transactions, however [12]. Timed order fairness assumes that all processes have access to synchronized local clocks; it can ensure that if all correct processes saw

message m to be ordered before m', then m is scheduled and delivered before m'. Similarly, ordering linearizability says that if the highest timestamp provided by any correct process for a message m is lower than the lowest timestamp provided by any correct process for a message m', then m will appear before m' in the output sequence. The implementation of ordering linearizability [19] uses a median computation, which can easily be manipulated by faulty processes [12].

The Hashgraph [3] consensus protocol also claims to achieve fairness. It uses gossip internally and all processes build a *hash graph* reflecting all of the gossip events. However, there is no formal definition of fairness and the presentation fails to recognize the impossibility of fair message-order resulting from the *Condorcet paradox*. Kelkar *et al.* [14] also show an attack that allows a malicious process to control the order of the messages delivered by Hashgraph.

A complementary measure to prevent message-reordering attacks relies on threshold cryptography [18, 5, 8]: clients encrypt their input (payload) messages under a key shared by the group of processes running the atomic broadcast protocol. They initially order the encrypted messages and subsequently collaborate for decrypting them. Hence, their contents become known only *after* the message order has been decided. For instance, the Helix protocol [2] implements this approach and additionally exploits in-protocol randomness for two additional goals: to elect the processes running the protocol from a larger group and to determine which messages among all available ones must be included by a process when proposing a block. This method provides resistance to censorship but still permits some order-manipulation attacks.

3 System model and preliminaries

System model. We model our system as a set of n *processes* $\mathcal{P} = \{p_1, \ldots, p_n\}$, also called *parties*, that communicate with each other. Processes interact with each other by exchanging messages reliably in a network. A protocol for $\mathcal{P}$ consists of a collection of programs with instructions for all processes. Processes are computationally bounded and protocols may use cryptographic primitives, in particular, digital signature schemes.

In our model, we distinguish two types of processes. Processes that follow the protocol as expected are called *correct*. Contrary, the processes that deviate from the protocol specification or may crash are called *Byzantine*.

We assume that there exists a low-level mechanism for sending messages over reliable and authenticated point-to-point links between processes. In our protocol implementation, we describe this as "sending a message" and "receiving a message". Additionally, we assume *first-in first-out (FIFO) ordering* for the links. This ensures that messages broadcast by the same correct process are delivered in the order in which they were sent by a correct recipient.

This work considers two models, *asynchrony* and *partial synchrony*. Together they cover most scenarios used today in the context of secure distributed computing. In an *asynchronous* network, no physical clock is available to any process and the delivery of messages may be delayed arbitrarily. In such networks, it is

only guaranteed that a message sent by a correct process will *eventually* arrive at its destination. One can define asynchronous time based on logical clocks. A *partially synchronous* network [9] operates asynchronously until some point in time (not known to the processes), after which it becomes stable. This means that processing times and message delays are bounded afterwards, but the maximal delays are not known to the protocol.

Broadcast and consensus primitives. The following primitives are important for our work: Byzantine FIFO consistent broadcast channel (BCCH), validated Byzantine consensus (VBC) and atomic broadcast.

BCCH allows the processes to deliver multiple payloads and ensures FIFO delivery and consistency despite Byzantine senders [4, Sec. 3.12]. BCCH provides two events: *bcch-broadcast*(m) and *bcch-deliver*(p_j, l, m).

Definition 1 (Byzantine FIFO Consistent Broadcast Channel). *A Byzantine FIFO consistent broadcast channel satisfies the following properties:*

Validity: *If a correct process broadcasts a message m, then every correct process eventually delivers m.*
No duplication: *For every process p_j and label l, every correct process delivers at most one message with label l and sender p_j.*
Integrity: *If some correct process delivers a message m with sender p_j and process p_j is correct, then m was previously broadcast by p_j.*
Consistency: *If some correct process delivers a message m with label l and sender p_j, and another correct process delivers a message m' with label l and sender p_j, then $m = m'$.*
FIFO delivery: *If a correct process broadcasts some message m before it broadcasts a message m', then no correct process delivers m' unless it has already delivered m.*

VBC [5] defines an *external validity* condition. It requires that the consensus value is legal according to a global, efficiently computable predicate P, known to all processes. A consensus primitive is accessed through the events *vbc-propose*(v) and *vbc-decide*(v).

Definition 2 (Validated Byzantine Consensus). *A protocol solves validated Byzantine consensus with validity predicate P if it satisfies the following conditions:*

Termination: *Every correct process eventually decides some value.*
Integrity: *No correct process decides twice.*
Agreement: *No two correct processes decide differently.*
External validity: *Every correct process only decides a value v such that $P(v) =$* TRUE. *Moreover, if all processes are correct and propose v, then no correct process decides a value different from v.*

Atomic broadcast ensures that all processes deliver the same messages and that all messages are output in the same order. This is equivalent to the processes

agreeing on one sequence of messages that they deliver. Processes may broadcast a message m by invoking *a-broadcast*(m), and the protocol outputs messages through *a-deliver*(m) events.

Definition 3 (Atomic Broadcast). *A protocol for atomic broadcast satisfies the following properties:*

Validity: *If a correct process* a-broadcasts *a message m, then every correct process eventually* a-delivers *m.*

No duplication: *No message is* a-delivered *more than once.*

Agreement: *If a message m is* a-delivered *by some correct process, then m is eventually* a-delivered *by every correct process.*

Total order: *Let m and m' be two messages such that p_i and p_j are correct processes that* a-deliver *m and m'. If p_i* a-delivers *m before m', then p_j also* a-delivers *m before m'.*

4 Revisiting order fairness

4.1 Limitations

Defining a fair order for atomic broadcast in asynchronous networks is not straightforward since the processes might locally receive messages for broadcasting in different orders. We assume here that a correct process receives a payload to be broadcast (e.g., from a client) at the same time when it *a-broadcasts* it. If a process broadcasts a payload message m before a payload message m', according to its local order, we denote this by $m \prec m'$.

Even if all processes are correct, it can be impossible to define a fair order among all messages. This is shown by a result from social science, known as the *Condorcet paradox*, which states that there exist situations that lead to non-transitive collective voting preferences even if the individual preferences are transitive. Kelkar *et al.* [14] apply this to atomic broadcast and show that delivering messages in a fair order is not always possible. Their example considers three correct processes p_1, p_2, and p_3 that receive three payload messages m_a, m_b, and m_c. While p_1 receives these payload messages in the order $m_a \prec m_b \prec m_c$, process p_2 receives them as $m_b \prec m_c \prec m_a$ and p_3 in the order $m_c \prec m_a \prec m_b$. Obviously, a majority of the processes received m_a before m_b, m_b before m_c, but also m_c before m_a, leading to a cyclic order. Consequently, a fair order cannot be specified even with only correct processes.

One way to handle situations with such cycles in the order is presented by Kelkar *et al.* [14] with *block-order fairness*: their protocol delivers a "block" of payload messages at once. Typically, a block will contain those payloads that are involved in a cyclic order. Their notion requires that if sufficiently many processes receive a payload m before another payload m', then no correct process delivers m after m', but they may both appear in the same block. Even though the order among the messages within a block remains unspecified, the notion of block-order fairness respects a fair order up to this limit.

Kelkar *et al.* [14] specify "sufficiently many" as a γ-fraction of *all* processes, where γ represents an order-fairness parameter such that $\frac{1}{2} < \gamma \leq 1$. More precisely, block-order fairness considers a number of processes η that all receive (and broadcast) two payload messages m and m'. Block-order fairness for atomic broadcast requires that whenever there are at least $\gamma\eta$ processes that receive m before m', then no correct process delivers m after m' (but they may deliver m and m' in the same block).

Kelkar *et al.* [14] explicitly count faulty processes for their definition. Notice that this immediately leads to problems: If $\gamma\eta < 2f$, for instance, the notion relies on a majority of faulty processes, but no guarantees are possible in this case. Therefore, we only count on events occurring at correct processes here and define a block-order fairness parameter $\overline{\gamma}$ to denote the fraction of *correct* processes that receive one message before the other.

Moreover, we assume w.l.o.g. that all correct processes eventually broadcast every payload, even if this is initially input by a single process only. This simplifies the treatment compared to original block-order fairness, which considers only processes that broadcast *both* payload messages, m and m' [14]. Our simplification means that a correct process that has received only one payload will receive the other payload as well later. This process should eventually include also the second payload for establishing a fair order. It corresponds to how atomic broadcast is used in practice; hence, we set $\eta = n - f$. In asynchronous networks, furthermore, one has to respect f additional correct processes that may be delayed. Their absence reduces the strength of the formal notion of block-order fairness in asynchronous networks even more.

In the following, we discuss the range of achievable values for $\overline{\gamma}$. Since we focus on models that allow asynchrony, we assume $n > 3f$ throughout this work. Fundamental results on validity notions for Byzantine consensus in asynchronous networks have been obtained by Fitzi and Garay [10]. Recall that a consensus protocol satisfies *termination*, *integrity*, and *agreement* according to Definition 2. *Standard consensus* additionally satisfies:

Validity: If all correct processes propose v, then all correct processes decide v.

Notice that this leaves the decision value completely open if only one correct process proposes something different. In their notion of *strong consensus*, however, the values proposed by correct processes must be better respected, under more circumstances:

Strong validity: If a correct process decides v, then some correct process has proposed v.

Unfortunately, strong consensus is not suitable for practical purposes because Fitzi and Garay [10, Thm. 8] also show that if the proposal values are taken from a domain $\mathcal{V}$, then the resilience depends on $|\mathcal{V}|$. In particular, strong consensus is only possible if $n > |\mathcal{V}|f$.

Related to this, they also introduce δ-*differential consensus*, which respects how many times a value is proposed by the correct processes. This notion ensures,

in short, that the decision value has been proposed by "sufficiently many" correct processes compared to how many processes proposed some different value. More precisely, for an execution of consensus and any value $v \in \mathcal{V}$, let $c(v)$ denote the number of correct processes that propose v:

δ-differential validity: If a correct process decides v, then every other value w proposed by some correct process satisfies $c(w) \leq c(v) + \delta$.

To summarize, whereas the standard notion of Byzantine consensus requires that *all* correct processes start with the same value in order to decide on one of the correct processes' input, strong consensus achieves this in any case. It requires that the decision value has been proposed by *some* correct process. However, it does not connect the decision value to how many correct processes have proposed it. Consequently, strong consensus may decide a value proposed by just one correct process. Differential consensus, finally, makes the initial plurality of the decision value explicit. For $\delta = 0$, in particular, the decision value must be one of the proposed values that is most common among the correct processes. More importantly, differential validity can be achieved under the usual assumption that $n > 3f$.

We now give another characterization of δ-differential validity. For a particular execution of some (asynchronous) Byzantine consensus protocol, let v^* be (one of) the value(s) proposed most often by correct processes, i.e., $v^* = \arg\max_v c(v)$.

Lemma 1. *A Byzantine consensus protocol satisfies δ-differential validity if and only if in every one of its executions, it never decides a value w with $c(w) < c(v^*) - \delta$.*

Proof. Assume first that the protocol satisfies δ-differential validity and a correct process decides any value v in the domain. Then every other value w proposed by a correct process satisfies $c(w) \leq c(v) + \delta$. In particular, this implies $c(v^*) \leq c(v) + \delta$, which is equivalent to, $c(v) \geq c(v^*) - \delta$. Hence, the protocol *never* decides a value x with $c(x) < c(v^*) - \delta$.

To show the reverse direction, suppose x is such that $c(x) < c(v^*) - \delta$ and a correct process decides x. This does not satisfy δ-differential validity because also v^* has been proposed by a correct process but $c(v^*) > c(x) + \delta$.

For consensus with a *binary* domain $\mathcal{V} = \{0, 1\}$, this means that a consensus protocol satisfies δ-differential validity if and only if in every one of its executions such that $c(0) > c(1) + \delta$, every correct process decides 0.

No asynchronous consensus algorithm for agreeing on the value proposed by a simple majority of correct processes exists, however. Fitzi and Garay [10, Thm. 11] prove that δ-differential consensus in asynchronous networks is *not possible* for $\delta < 2f$:

Theorem 1 ([10]). *In an asynchronous network, δ-differential consensus is achievable only if $\delta \geq 2f$.*

The above discussion already hints at issues with achieving fair order in asynchronous systems. Recall that Kelkar *et al.* [14] present atomic broadcast protocols with block-order fairness for the asynchronous setting with order-fairness parameter γ (whose definition includes faulty processes). The corruption bound is stated as $n > \frac{4f}{2\gamma-1}$. For $\gamma = 1$, which ensures fairness only in the most clear cases, there are $n > 4f$ processes required. For values of γ close to $\frac{1}{2}$, the condition becomes prohibitive for practical solutions. In fact, even when using our interpretation, $\overline{\gamma}$ cannot be too close to $\frac{1}{2}$, as the following result shows. It rules out the existence of $\overline{\gamma}$-block-order-fair atomic broadcast in asynchronous or eventually synchronous networks for $\overline{\gamma} < \frac{1}{2} + \frac{f}{n-f}$.

Theorem 2. *In an asynchronous network with n processes and f faults, implementing atomic broadcast with $\overline{\gamma}$-fair block order is not possible unless $\overline{\gamma} \geq \frac{1}{2} + \frac{f}{n-f}$.*

Proof. Towards a contradiction, suppose there is an atomic broadcast protocol ensuring $\overline{\gamma}$-fair block order with $\frac{1}{2} < \overline{\gamma} < \frac{1}{2} + \frac{f}{n-f}$. We will transform this into a differential consensus protocol that violates Theorem 1.

The consensus protocol works like this. All processes initialize the atomic broadcast protocol. Upon *propose*(v) with some value v, a process simply *a-broadcasts* v. When the first value v' is *a-delivered* by atomic broadcast to a process, the process executes *decide*(v') and terminates.

Consider any execution of this protocol such that all correct processes propose one of two values, m or m'. Suppose w.l.o.g. that $c(m) = \overline{\gamma}(n-f)$ and $c(m') = (1-\overline{\gamma})(n-f)$, i.e., m is proposed $c(m)$ times by correct processes and more often than m', since $\overline{\gamma} > \frac{1}{2}$. It follows that $\overline{\gamma}(n-f)$ correct processes *a-broadcast* m before m' and $(1-\overline{\gamma})(n-f)$ correct processes *a-broadcast* m' before m.

According to the properties of atomic broadcast all correct processes *a-deliver* the same value first in every execution. Moreover, the atomic broadcast protocol *a-delivers* m before m' by the $\overline{\gamma}$-fair block order property. This implies that the consensus protocol decides m in every execution and never m'. Since no further restrictions are placed on m and on m', this consensus protocol actually ensures δ-differential validity for some $\delta < c(m) - c(m')$ by Lemma 1.

However, the $c(m)$ and $c(m')$ satisfy, respectively,

$$\begin{aligned} c(m) &= \quad \overline{\gamma}(n-f) \quad < \quad \left(\tfrac{1}{2} + \tfrac{f}{n-f}\right)(n-f) \quad = \tfrac{n+f}{2} \\ c(m') &= (1-\overline{\gamma})(n-f) > \left(1 - \tfrac{1}{2} - \tfrac{f}{n-f}\right)(n-f) = \tfrac{n-3f}{2} \end{aligned}$$

and, therefore, $\delta < c(m) - c(m') < \frac{n+f}{2} - \frac{n-3f}{2} = 2f$. But δ-differential asynchronous consensus is only possible when $\delta \geq 2f$, a contradiction.

4.2 Differential Order-Fairness

The limitations discussed above have an influence on order fairness. The condition on δ to achieve *δ-differential consensus* directly impacts any measure of

fairness. It becomes clear that a *relative* notion for block-order fairness, defined through a fraction like $\overline{\gamma}$, may not be expressive enough.

We now start to define our notion of *order-fair atomic broadcast*; it has almost the same interface as regular atomic broadcast. The primitive is accessed with *of-broadcast*(m) for broadcasting a payload message m and it outputs payload messages through *of-deliver*(M) events, where M is a *set* of payloads delivered at the same time; M corresponds the block of block-order fairness. We want to count the number of correct processes that *of-broadcast* a message m before another message m' and introduce a function $b : \mathcal{M} \times \mathcal{M} \to \mathbb{N}$ for all m and m' that were ever *of-broadcast* by correct processes. The value $b(m, m')$ denotes the *number of correct processes* that *of-broadcast* m before m' in an execution. As above we assume w.l.o.g. that a correct process will *of-broadcast* m and m' eventually and that, therefore, $b(m, m') + b(m', m) = n - f$.

Can we achieve that if $b(m, m') > b(m', m)$, i.e., when there are more correct processes that *of-broadcast* message m before m' than correct processes that *of-broadcast* m' before m, then no correct process will *of-deliver* m' before m? Using a reduction from δ-differential consensus, as in the previous result, we can show that this condition is too weak. The proof is provided in the full version [6].

Theorem 3. *Consider an atomic broadcast protocol that satisfies the following notion of order fairness for some $\mu \geq 0$:*

Weak differential order fairness: *For any m and m', if $b(m, m') > b(m', m) + \mu$, then no correct process a-delivers m' before m.*

Then it must hold $\mu \geq 2f$.

On the basis of this result, we now formulate our notion of *κ-differentially order-fair atomic broadcast*, using a fairness parameter $\kappa \geq 0$ to express the strength of the fairness. Smaller values of κ ensure stronger fairness in the sense of how large the majority of processes that *of-broadcast* some m before m' must be to ensure that m will be *of-delivered* before m' and in a fair order. We strengthen the *validity* property of the regular definition of atomic broadcast (Definition 3) to *strong validity* such that we only include payload messages in a fair order that were *of-broadcast* by more than f correct processes.

Definition 4 (κ-Differentially Order-Fair Atomic Broadcast). *A protocol for* κ-differentially order-fair atomic broadcast *satisfies the properties* no duplication, agreement *and* total order *of* atomic broadcast *and additionally:*

Strong validity: *If* more than f *correct processes of-broadcast a message m, then every correct process eventually of-delivers m.*

κ-differential order fairness: *If $b(m, m') > b(m', m) + 2f + \kappa$, then no correct process of-delivers m' before m.*

Compared to the above notion of weak differential order fairness, we have $\kappa = \mu - 2f$. We show in the next section how to implement κ-differentially order-fair atomic broadcast.

5 Quick order-fair atomic broadcast protocol

5.1 Overview

The protocol concurrently runs a Byzantine FIFO consistent broadcast channel (BCCH) and proceeds in rounds of consensus. BCCH allows processes to deliver multiple messages consistently. An incoming *of-broadcast* event with a payload message m triggers BCCH and *bcch-broadcasts* m to the network. Additionally, every process keeps a local vector clock that counts the payloads that have been *bcch-delivered* from each sending process. Every process also maintains an array of lists *msgs* such that $msgs[i]$ records all *bcch-delivered* payloads from p_i.

When a process *bcch-delivers* the payload message m, it increments the corresponding vector-clock entry and appends m to the appropriate list in *msgs*. As soon as sufficiently many new payloads are found in *msgs*, a new round starts. Each process signs its vector clock and sends it to all others. The received vector clocks are collected in a matrix, and once $n - f$ valid vector clocks are recorded, a new validated Byzantine consensus (VBC) instance is triggered. The process proposes the matrix and the signatures for consensus, and VBC decides on a common matrix with valid signatures. This matrix defines a *cut*, which is a vector of indices, with one index per process, such that the index for p_j determines an entry in $msgs[j]$ up to which payload messages are considered for creating the fair order in the round. It may be that the index points to messages that a process p_i does not store in $msgs[j]$ because they have not been *bcch-delivered* yet. When the process detects such a missing payload, it asks all other processes to send the missing payload directly and in a verifiable way, such that every process will store all payloads up to the cut in *msgs*.

Once all processes received the payloads up to the cut, the algorithm starts to build a graph that represents the dependencies among messages that must be respected for a fair order. This graph resembles the one used in Aequitas [14], but its semantics and implementation differ. The vertices in the graph here are all *new* payload messages defined by the cut and an edge (m, m') indicates that m should at most be *of-delivered* before m'.

The graph results from two steps. In the first step, the process creates a vertex for every payload message that appears in more than f distinct lists in *msgs*. In the second step, the algorithm builds a matrix M such that $M[m][m']$ counts how many times m appears before m' in *msgs* (up to the cut). $M[m][m']$ can be interpreted as *votes*, counting how many processes want to order m before m'. Notice that entries of M exist only for m and m' where at least one of $M[m][m']$ and $M[m'][m]$ is non-zero.

If the *difference* between entries $M[m][m']$ and $M[m'][m]$ is large enough, then the protocol adds a directed edge (m, m') to the graph. The edge indicates that m' must not be *of-delivered* before m. More precisely, such an edge is added for all m and m' with $M[m][m'] > M[m'][m] - f + \kappa$. The condition is explained through the following result.

Lemma 2. *If* $b(m, m') > b(m', m) + 2f + \kappa$, *then* $M[m][m'] > M[m'][m] - f + \kappa$.

Proof. At least $M[m][m'] - f$ correct processes have *of-broadcast* m before m' because $M[m][m']$ may include reports about m and m' in *msgs* from up to f incorrect processes. In other words,

$$b(m, m') \geq M[m][m'] - f \quad \Longleftrightarrow \quad M[m][m'] \leq b(m, m') + f$$

At most $M[m][m'] + 2f$ correct processes have *of-broadcast* m before m' because $M[m][m']$ may include reports about m or m' in *msgs* from up to f incorrect processes, and there may be up to $2f$ correct processes whose arrays were not considered in this number. That is,

$$b(m, m') \leq M[m][m'] + 2f \quad \Longleftrightarrow \quad M[m][m'] \geq b(m, m') - 2f$$

Suppose $b(m, m') > b(m', m) + 2f + \kappa$. The above implies

$$\begin{aligned} M[m][m'] &\geq b(m, m') - 2f > b(m', m) + 2f + \kappa - 2f = b(m', m) + \kappa \\ &\geq M[m'][m] - f + \kappa. \end{aligned}$$

Thus, whenever $M[m][m'] > M[m'][m] - f + \kappa$, we need to prevent that the protocol *of-delivers* m' before m. We do this by adding an edge from m to m' to the graph; as shown later, this ensures that m' is not *of-delivered* before m.

Creating the graph in this manner leads to a directed graph that represents constraints to be respected by a fair order. Notice that two messages may be connected by edges in both directions when the difference is small and $\kappa < f$, i.e., there may be a cycle (m, m') and (m', m). This means that the difference between the number of processes voting for one or the other order is too small to decide on a fair order. Longer cycles may also exist. Note that a cycle length cannot be longer than the number of messages, which correspond to nodes in the graph. All payload messages with circular dependencies among them will be *of-delivered* together as a set. For deriving this information, the algorithm repeatedly detects all strongly connected components in the graph and collapses them to a vertex. In other words, any two vertices m and m' are merged when there exists a path from m to m' and a path from m' to m. This technique also handles cases like those derived from the *Condorcet paradox*.

Finally, with the help of the collapsed graph, all payload messages defined by the cut are *of-delivered* in a fair order: First, all vertices without any incoming edges are selected. Secondly, these vertices are sorted in a deterministic way and the corresponding payloads are *of-delivered* one after the other. Then the processed vertices are removed from the graph and another iteration through the graph starts. As soon as there are no vertices left, i.e. all payload messages are *of-delivered*, the protocol proceeds to the next round.

5.2 Implementation

Algorithm 1–2 shows the *quick order-fair atomic broadcast protocol* for a process p_i. The protocol proceeds in rounds and maintains a round counter r (L1)

and uses a boolean variable *inround*, which indicates whether the consensus phase of a round is executing (L2).

Every process maintains two hash maps: *msgs* (L3) and *vc* (L4). The process identifiers serve as keys in both hash maps. Hash map *msgs* contains ordered lists of *bcch-delivered* payload from each process in the system. Variable *vc* is a vector clock counting how many payload messages were *bcch-delivered* from each process.

Rounds. In each round, a matrix L (L5) and a list Σ (L6) are constructed as inputs for consensus. The matrix L will consist of vector clocks from the processes and Σ will contain the signatures of the processes. Additionally, every process maintains a list of values called *cut* (L7) that are calculated in every round. This cut represents an index for every list in *msgs* to determine the payload to be used for creating the fair order. Initially, all values are zero. Finally, all *of-delivered* payload messages are included in a set *delivered* (L8), to prevent a repeated delivery in future rounds.

The protocol starts when a client submits a payload message m using an *of-broadcast*(m) event. BCCH then broadcasts m to all processes in the network (L11). When m with label l from process p_j is *bcch-delivered* (L12), the vector clock *vc* for process p_j is incremented. The attached label l is not used by the algorithm and only serves to define that all correct processes *bcch-deliver* the same payload following Definition 1. Additionally, payload m is appended to the list *msgs*[j] using an operation *append*(m) (L14).

When the length of p_j's list in *msgs* exceeds the *cut* value for p_j, new payloads may have arrived that should be ordered (L15). This tells the protocol to initiate a new round. This condition can be adapted as described in the remarks at the end of this section.

The first step of round r is to set the flag *inround*. Secondly, the protocol digitally signs the vector clock *vc* and obtains a signature σ. The values r, σ, and *vc* are then sent in a STATUS message to all processes (L16–L18). When process p_i receives a STATUS message from p_j, it validates the contained signature σ' using *verify*(j, vc', σ') (L20). An additional security check is made by comparing the locally stored round number r with the round number r' from the message. If both conditions hold, the vector clock vc' is stored as row j in matrix L (L21) and σ' is stored in list Σ at index j (L22).

Defining a cut. As soon as p_i has received $n - f$ valid STATUS-messages (L23), it invokes consensus (VBC, L24) for the round through *vbc-propose* with proposal (L, Σ). When the VBC protocol subsequently decides, it outputs a common matrix L' of vector clocks and a list Σ' of signatures (L26). The process then uses L' to calculate the cut, where *cut*[j] is the largest value s such that at least $f + 1$ elements in column j in L' are bigger or equal than s (L29). In other words, *cut*[j] represents how many payload messages from p_j were *bcch-delivered* by enough processes. This value is used as index into *msgs*[j] to determine the payloads that will be considered for creating the order in this round.

The algorithm then makes sure that all processes will hold at least all those payloads in *msgs* that are defined by *cut*. Each process detects missing payload messages from sender p_j from any difference between $vc[j]$ and $cut[j]$ (L31); if there are any, the process broadcasts a MISSING-message to all others. When another process receives such a request from p_j and already has the requested payloads in *msgs*, it extracts them into a variable *resend* (L36). More precisely, it extracts a proof from the BCCH primitive with which any other process can verify that the payload from this particular sender is genuine. This is done by invoking *bcch-create-proof*(*resend*) (L37); the messages and the proof are then sent in a RESEND-message to the requesting process p_j (L38).

When process p_i receives a RESEND-message with a missing payload from p_k, it verifies the provided proof from the message by invoking a *bcch-verify-proof*(s') function (L41). If the proof is valid, p_i extracts (L43) the payload messages through *bcch-get-messages*(s'), appends them to $msgs[k]$, and increments $vc[k]$ accordingly. The process repeats this until *msgs* contains all payloads included in the cut.

Ordering messages. At this point, every process stores all payload messages *msgs* that have been *bcch-delivered* up to the cut. The remaining operations of the round are deterministic and executed by all processes independently.

The next step is to construct the directed *dependency graph* G that expresses the constraints on the fair order of the payload messages. Vertices in G represent payload messages that will be *of-delivered* and edges in G express constraints on the order among these payloads. A message is only added to G when it has not yet been *of-delivered*, when it is contained inside the *cut*, and when it appears in more than f lists in *msgs* (L46–L49). The latter condition will guarantee that a payload is only *of-delivered* if it was *bcch-broadcast* by at least one correct process.

The algorithm then constructs M (L52) such that $M[m][m']$ counts how many times a payload m appears before payload m' in any list of *msgs*. Counting considers only m and m' that are vertices in G, but when some m appears in a list $msgs[j]$ and m' does not (because the process has not yet *bcch-delivered* m' from p_j), we assume that also m' will be *bcch-delivered* eventually, and hence count m before m'. Finally, all entries $M[m][m']$ and $M[m'][m]$ are compared and if $M[m][m'] > M[m'][m] - f + \kappa$, then a directed edge from m to m' is added (L53). This edge indicates that m must not be ordered *after* m', i.e., that m is *of-delivered* before m' or together with m'.

Any payloads that cannot be ordered with respect to each other now correspond to strongly connected components of G. (A strongly connected component is a subgraph, which for each pair of vertices m and m' contains a path from m to m' and one from m' to m.) In the next step, a graph $H = (W, F)$ is created and all strongly connected components in H are repeatedly collapsed until H contains no more cycles. This is done by contracting the edges in each connected component and merging all its vertices (L54–L56). The resulting graph H can then be traversed according this partial order. The algorithm considers all vertices $\bar{w}$ without an incoming edge: this indicates that there is no other vertex

Algorithm 1 Quick order-fair atomic broadcast (code for p_i).

State

$r \leftarrow 1$: current round
$\mathit{inround} \leftarrow$ FALSE
$\mathit{msgs} \leftarrow [\,] : \text{HashMap}\big[\{1, ..., n\} \to [\,]\big]$: lists of *bcch-delivered* messages
$\mathit{vc} \leftarrow [\,] : \text{HashMap}\big[\{1, ..., n\} \to \mathbb{N}\big]$: counters for *bcch*-delivered messages
$L \leftarrow [0]^{n \times n}$: matrix of logical timestamps
$\Sigma \leftarrow [\,]^n$: list of signatures from STATUS messages
$\mathit{cut} \leftarrow [0]^n$: the cut decided for the round
$\mathit{delivered} \leftarrow \emptyset$: set of delivered messages

Initialization

Byzantine FIFO consistent broadcast channel (*bcch*)

upon *of-broadcast*(m) **do**
    bcch-broadcast(m)

upon *bcch-deliver*(p_j, l, m) **do**
    $\mathit{vc}[j] \leftarrow \mathit{vc}[j] + 1$
    $\mathit{msgs}[j].\mathit{append}(m)$

upon exists j **such that** $\mathit{len}(\mathit{msgs}[j]) > \mathit{cut}[j] \wedge \neg \mathit{inround}$ **do**
    $\mathit{inround} \leftarrow$ TRUE
    $\sigma \leftarrow \mathit{sign}(i, \mathit{vc})$
    send message $[\text{STATUS}, r, \mathit{vc}, \sigma]$ to all $p_j \in \mathcal{P}$

upon receiving message $[\text{STATUS}, r', \mathit{vc}', \sigma']$ from p_j
    such that $r' = r \wedge \mathit{verify}(j, \mathit{vc}', \sigma')$ **do**
        $L[j] \leftarrow \mathit{vc}'$
        $\Sigma[j] \leftarrow \sigma'$

upon $\big|\{p_j \in \mathcal{P} \mid \Sigma[j] \neq \perp\}\big| \geq n - f$ **do**
    vbc-propose$\big((L, \Sigma)\big)$ for validated Byzantine consensus in round r
    $\Sigma \leftarrow [\,]^n$

upon *vbc-decide*$\big((L', \Sigma')\big)$ in round r **do** // calculate cut
    for $j \in \{1, \ldots, n\}$ **do** // for each row in L'
        // $\mathit{cut}[j]$ is largest s s.t. at least $f + 1$ el. in col. j in L' are at least s
        $\mathit{cut}[j] \leftarrow \mathit{max}\big\{s \mid \{k \mid \big|\{L'[k][j] \geq s\}\big| > f\}\big\}$

    for $j \in \{1, \ldots, n\}$ **do** // check for missing messages
        if $\mathit{vc}[j] < \mathit{cut}[j]$ **then**
            send message $[\text{MISSING}, r, j, \mathit{vc}[j]]$ to all $p_k \in \mathcal{P}$

upon receiving message $[\text{MISSING}, r', k, \mathit{index}]$ from p_j
    such that $r' = r$ **do**
        if $\mathit{vc}[k] \geq \mathit{cut}[k]$ **then**
            $\mathit{resend} \leftarrow \mathit{msgs}[k].\mathit{get}(\mathit{index} \ldots \mathit{cut}[k])$ // copy messages from p_k
            $s \leftarrow$ *bcch-create-proof*(resend)
            send message $[\text{RESEND}, r, k, s]$ to p_j // send missing messages

Algorithm 2 Quick order-fair atomic broadcast (code for p_i).

39: **upon** receiving message $[\text{RESEND}, r', k', s']$ from p_j
40: **such that** $r' = r \wedge \mathit{len}(\mathit{msgs}[k]) < \mathit{cut}[k]$ **do**
41: **if** *bcch-verify-proof*(s') **then**
42: $\mathit{vc}[k] \leftarrow \mathit{vc}[k] + \mathit{bcch\text{-}get\text{-}length}(s')$
43: $\mathit{msgs}[k].\mathit{append}(\mathit{bcch\text{-}get\text{-}messages}(s'))$

44: **upon** $\mathit{len}(\mathit{msgs}[j]) \geq \mathit{cut}[j]$ for all $j \in \{1, \ldots, n\}$ **do**
45: $G \leftarrow (\emptyset, \emptyset)$ // $G = (V, E)$
46: **for** $m \in \left(\bigcup_{j \in \{1,\ldots,n\}} \mathit{msgs}[j]\right) \setminus \mathit{delivered}$ **do**
47: // add vertex for any m that appears in more than f lists in *msgs*
48: **if** $\left|\{j | m \in \mathit{msgs}[j].\mathit{get}(1 \ldots \mathit{cut}[j])\}\right| > f$ **then**
49: $V \leftarrow V \cup \{m\}$
50: $M \leftarrow [\,]$: HashMap$[\mathcal{M} \times \mathcal{M} \to \mathbb{N}]$
51: **for** $m, m' \in V$ **do** // build matrix with votes
52: $M[m][m'] \leftarrow \left|\{j \in \{1, \ldots, n\} \mid m \text{ appears before } m' \text{ in } \mathit{msgs}[j]\}\right|$
53: $E \leftarrow \{(m, m') \mid M[m][m'] > M[m'][m] - f + \kappa\}$ // edge when diff. small
54: $H \leftarrow G$
55: **while** H contains strongly connected subgraph $\overline{H} = (\overline{W}, \overline{F}) \subseteq H$ **do**
56: $H \leftarrow H/\overline{F}$ // collapse vertices via edge contraction
57: // $H = (W, F)$
58: **while** $|W| > 0$ **do**
59: $\mathit{sources} \leftarrow \{\bar{w} \in W \mid \mathit{indegree}(\bar{w}) = 0\}$ // vertices w.o. incoming edge
60: **for** $\bar{w} \in \mathit{sort}(\mathit{sources})$ **do** // $\bar{w}$ is one message or a set of messages
61: **if** $|\bar{w}| = 1$ **then**
62: *of-deliver*$(\bar{w})$ // output a single message
63: $\mathit{delivered} \leftarrow \mathit{delivered} \cup \{\bar{w}\}$ // keep track of del. messages
64: **else**
65: *of-deliver*$(\bar{w})$ // output a set (block) of messages
66: $\mathit{delivered} \leftarrow \mathit{delivered} \cup \bar{w}$ // keep track of del. messages
67: $H \leftarrow H \setminus \bar{w}$ // remove a vertex from $H = (W, F)$
68: $L \leftarrow [0]^{n \times n}$
69: *inround* $\leftarrow$ FALSE
70: $r \leftarrow r + 1$ // move to the next round

that must be ordered before $\bar{w}$. All such $\bar{w}$ will be collected (in a set *sources*), sorted in a deterministic way, and *of-delivered* in this order (L59, L60, L62, L65). Notice that $\bar{w}$ may correspond to a set of payload messages resulting from collapsing a subgraph: they are delivered together as a set. All *of-delivered* payload messages are also added to *delivered* (L63, L66) to prevent a repeated processing. Finally, $\bar{w}$ is removed from H (L67), and a next pass of extracting vertices with no incoming edge follows. This is repeated until all vertices have been processed and *of-delivered*. The algorithm then initializes L, increments the round number r, and starts the next round (L68-L70).

6 Complexity

If the Byzantine FIFO consistent broadcast channel (BCCH) is implemented using "echo broadcast" [17], it takes $O(n)$ protocol messages per payload message. Since more than f processes *of-broadcast* each payload message and f is proportional to n, the overall message complexity of BCCH is $O(n^2)$. The cost of validated Byzantine consensus (VBC) depends on the assumptions used for implementing it. With a partially synchronous consensus protocol VBC uses $O(n)$ messages in the best case and $O(n^2)$ messages in the worst case. The total amortized cost of quick order-fair atomic broadcast per payload, therefore, is also $O(n^2)$ messages in this implementation. If digital signatures are of length λ and payload messages are at most L bits, the bit complexity of BCCH for one sender is $O(n^2L+n^3\lambda)$. Optimal asynchronous VBC protocols [1, 16] have $O(nL+n^2\lambda)$ expected communication cost, for their payload length L. Since the proposals for VBC are $n \times n$ matrices, it follows that the amortized bit complexity of the algorithm per payload message is $O(n^2L + n^3\lambda)$.

Table 1 compares the cost of different order-fair atomic broadcast protocols. The asynchronous Aequitas protocol [14, Sec. 7] provides fair order using a *FIFO Broadcast primitive*, implemented by *OARcast* of Ho *et al.* [11]. Aequitas uses $\Omega(n^4)$ messages for delivering one payload, which exceeds the cost of quick order-fair broadcast at least by the factor n^2. The Pompē protocol cost is $O(n^2)$ messages and one instance of Byzantine consensus per payload message. The communication complexity of this protocol is $O(n^3L)$ since each process broadcasts a SEQUENCE-message to all others with contents of length $O(nL)$.

Notion	Algorithm	Avg. messages	Avg. comm.
Block-Order-Fairness [14]	Async. Aequitas [14]	$O(n^4)$	$O(n^4L)$
Ordering Linearizability [19]	Pompē* [19]	$O(n^2)$	$O(n^3L)$
Differential Order Fairness	Quick o.-f. broadcast	$O(n^2)$	$O(n^2L + n^3\lambda)$

Table 1. Expected message and communication complexities of different protocols, assuming $L \geq \lambda$. (* The Pompē protocol requires synchronized clocks.)

7 Conclusion

The quick order-fair atomic broadcast protocol guarantees payload message delivery in a differentially fair order. It works both for asynchronous and eventually synchronous networks with optimal resilience, tolerating corruptions of up to one third of the processes. Compared to existing order-fair atomic broadcast protocols, our protocol is considerably more efficient and incurs only quadratic cost in terms of amortized message complexity per delivered payload.

Acknowledgments

We thank the anonymous reviewers for helpful suggestions and feedback. This work has been funded by the Swiss National Science Foundation (SNSF) under grant agreement Nr. 200021_188443 (Advanced Consensus Protocols).

References

1. Abraham, I., Malkhi, D., Spiegelman, A.: Asymptotically optimal validated asynchronous byzantine agreement. In: PODC. pp. 337–346. ACM (2019)
2. Asayag, A., Cohen, G., Grayevsky, I., Leshkowitz, M., Rottenstreich, O., Tamari, R., Yakira, D.: A fair consensus protocol for transaction ordering. In: ICNP. pp. 55–65. IEEE Computer Society (2018)
3. Baird, L.: The Swirlds hashgraph consensus algorithm: Fair, fast, byzantine fault tolerance. Swirlds Tech Report, SWIRLDS-TR-2016-01 (2016), https://www.swirlds.com/downloads/SWIRLDS-TR-2016-01.pdf
4. Cachin, C., Guerraoui, R., Rodrigues, L.E.T.: Introduction to Reliable and Secure Distributed Programming (2. ed.). Springer (2011)
5. Cachin, C., Kursawe, K., Petzold, F., Shoup, V.: Secure and efficient asynchronous broadcast protocols. In: CRYPTO. Lecture Notes in Computer Science, vol. 2139, pp. 524–541. Springer (2001)
6. Cachin, C., Mićić, J., Steinhauer, N.: Quick Order Fairness (2021), https://arxiv.org/abs/2112.06615
7. Daian, P., Goldfeder, S., Kell, T., Li, Y., Zhao, X., Bentov, I., Breidenbach, L., Juels, A.: Flash boys 2.0: Frontrunning in decentralized exchanges, miner extractable value, and consensus instability. In: IEEE Symposium on Security and Privacy. pp. 910–927. IEEE (2020)
8. Duan, S., Reiter, M.K., Zhang, H.: Secure causal atomic broadcast, revisited. In: DSN. pp. 61–72. IEEE Computer Society (2017)
9. Dwork, C., Lynch, N.A., Stockmeyer, L.J.: Consensus in the presence of partial synchrony. J. ACM **35**(2), 288–323 (1988)
10. Fitzi, M., Garay, J.A.: Efficient player-optimal protocols for strong and differential consensus. In: PODC. pp. 211–220. ACM (2003)
11. Ho, C., Dolev, D., van Renesse, R.: Making distributed applications robust. In: OPODIS. Lecture Notes in Computer Science, vol. 4878, pp. 232–246. Springer (2007)
12. Kelkar, M., Deb, S., Kannan, S.: Order-fair consensus in the permissionless setting. IACR Cryptol. ePrint Arch. (139) (2021), https://eprint.iacr.org/2021/139
13. Kelkar, M., Deb, S., Long, S., Juels, A., Kannan, S.: Themis: Fast, strong order-fairness in byzantine consensus. IACR Cryptol. ePrint Arch. (1465) (2021), https://eprint.iacr.org/2021/1465
14. Kelkar, M., Zhang, F., Goldfeder, S., Juels, A.: Order-fairness for byzantine consensus. In: CRYPTO (3). Lecture Notes in Computer Science, vol. 12172, pp. 451–480. Springer (2020)
15. Kursawe, K.: Wendy, the good little fairness widget: Achieving order fairness for blockchains. In: AFT. pp. 25–36. ACM (2020)
16. Lu, Y., Lu, Z., Tang, Q., Wang, G.: Dumbo-mvba: Optimal multi-valued validated asynchronous byzantine agreement, revisited. In: PODC. pp. 129–138. ACM (2020)
17. Reiter, M.K.: Secure agreement protocols: Reliable and atomic group multicast in rampart. In: CCS. pp. 68–80. ACM (1994)
18. Reiter, M.K., Birman, K.P.: How to securely replicate services. ACM Trans. Program. Lang. Syst. **16**(3), 986–1009 (1994)
19. Zhang, Y., Setty, S.T.V., Chen, Q., Zhou, L., Alvisi, L.: Byzantine ordered consensus without byzantine oligarchy. In: OSDI. pp. 633–649. USENIX Association (2020)